

華中科技大學

課程實驗報告

課程名稱： 大數據算法綜合實踐

專業班級： 大數據 2101 班

學 號： U202115652

姓 名： 李嘉鵬

指導教師： 顧琳

報告日期： 2024 年 6 月 26 日

計算機科學與技術學院

目录

实验 1： 大规模图数据中三角形计数算法的设计与性能优化	3
1.1 实验概述	3
1.2 实验内容	3
1.2.1 阶段 1 数据集特点分析	4
1.2.2 阶段 2 算法基本实现思路	4
1.2.3 优化点 1：使用基于互斥锁的 OpenMP 多线程，并行读取并处理输入数据 ..	9
1.2.4 优化点 2：使用管道（Pipeline）进行通信优化，降低读写开销	10
1.2.5 优化点 3：使用__gnu_parallel::sort 优化算法进行并行排序	10
1.2.6 优化点 4：基于 OpenMP 的多线程并行三角形计数	10
1.2.7 算法流程图	11
1.2.8 阶段 3 实验结果与分析	11
1.3 实验小结	13

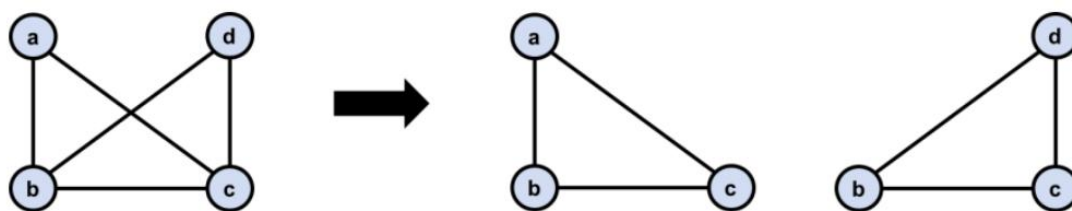
实验 1：大规模图数据中三角形计数算法的设计与性能优化

1.1 实验概述

本实验旨在通过大规模图数据中三角形计数算法的设计与性能优化，帮助学生深入理解图计算系统的工作原理和性能优化机制，并学会使用图计算框架进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本概念、编写比较复杂的图算法程序并进行性能调优。

1.2 实验内容

大数据时代，对关联（图）数据的处理被广泛应用于社交网络、智能交通、移动网络等领域。对图数据的三角形计数被广泛应用于图数据的特征描绘（如聚集系数、联通度等）、社区结构检索、子图匹配、生物网络等应用。本实验要求在给定服务器平台以及数据集上实现三角形计数（Triangle Counting, TC）算法，调试并获得最高的性能。三角形的定义是一个包含三个顶点的子图，其中顶点两两相连。例如，以下的无向图中包含 2 个三角形。



算法应着重讨论简单无向图的情形，即将重边（multi-edge）看成一条边，同时不考虑 loop（顶点指向自己的边）。如在上图中，若存在顶点 b 到 c 的两条边，则应忽略其中一条，这样，结果仍然是找到 2 个三角形。在本实验的数据集中，也应将有向图看成是无向图，需要重点考虑如何在大规模图数据中精确计算三角形的个数。在真实数据集中，图数据的规模将达到 $2^{28} \sim 2^{29}$ 顶点和 4-5 billion 条边，需要很长的计算时间。因此可以先构造比较简单的数据集用于测试代码功能。

本实验有功能和性能两方面的需求，首先需要能正确地统计出所给数据集中的所有三角形，其次需要不断优化算法（存储优化、通信优化），使得最终在测试机器上执行时间最短。

实验环境如下：

操作系统：Linux ubuntu 14.04 或 16.04

编译器：gcc 或者 g++ 4.8 以上

Make：GNU make 4.0 以上

框架：Spark GraphX、Pregel 或你所熟悉擅长的其它框架

1.2.1 阶段 1 数据集特点分析

- 1.任务描述：为了针对具体数据集实现三角形计数算法，首先需要分析给定的图数据集的格式。
- 2.实验过程：本实验共包含两个数据集，分别为大数据集和小数据集，如下表所示。

表 1：实验数据集说明

数据集	说明
soc-LiveJournal1	顶点数：4.8 million、边数：69 million；来源： https://snap.stanford.edu/data/soc-LiveJournal1.html
cit-HepPh	顶点数：34546、边数：421578；来源： https://snap.stanford.edu/data/cit-HepPh.html

打开小数据集 cit-HepPh，可以观察到其前四行为数据集信息（包括数据集的顶点总数和边的总数），后面每一行都包含两个被分隔符分开的数字，代表两个顶点的 id，如图 1 所示。

```
1 # Directed graph (each unordered pair of nodes is saved once): Cit-HepPh.txt
2 # Paper citation network of Arxiv High Energy Physics category
3 # Nodes: 34546 Edges: 421578
4 # FromNodeId ToNodeId
5 9907233 9301253
6 9907233 9504304
7 9907233 9505235
8 9907233 9506257
9 9907233 9606402
10 9907233 9607354
```

图 1：数据集特点分析

1.2.2 阶段 2 算法基本实现思路

- 1.任务描述：设计算法实现三角形计数。
- 2.实验设计：本次实验我采用 C++语言实现。算法主要分为以下 7 个步骤。

①读取数据集，构建边的列表

- a. 读取文件时需要跳过开头的四行无关注释信息；
- b. 使用 `vector<pair<ul,ul>> edges` 存储边的信息，其中每条边由两个顶点组成；
- c. 记录图的顶点数 `n` 和边数 `e`；在存储边的信息时确保每条边的第一个顶点 `id` 都小于第二个顶点 `id`，并且忽略自环，从而方便后续操作。

```

1.  ifstream file(filename);
2.  string tip;
3.  for (int i = 0; i < 4; ++i)
4.      getline(file, tip);
5.
6.  vector<pair<ul,ul>> edges;
7.  edges.reserve(100000000);
8.  ul n=0, u, v;
9.  intEdge e = 0;
10.
11. while(file >> u >> v) {
12.     e++;
13.     if(u == v) continue; // self-loop
14.     if(u > v) swap(u, v); // 确保 u <= v
15.     edges.push_back(make_pair(u, v));
16.     n = max3(n, u, v);
17. }
18. n++;
19. file.close();

```

②对边的列表进行排序

- 使用 GNU STL 的并行排序算法 `__gnu_parallel::sort` 对边列表进行排序（对应 1.2.4 节优化点 3）。
- 按照从小到大的顺序进行升序排序。

```

1.  __gnu_parallel::sort(edges.begin(), edges.end());

```

③删除重复的边

- 从排序后的边列表中删除重复的边，生成不重复的边列表。
- 在计算完成后立刻使用 `unique_edges` 保存不重复的边（对应 1.2.4 节优化点 2）；在判定一条边是否唯一时，同时记录唯一边的数量。

```

1.  vector<pair<ul, ul>> unique_edges;
2.  unique_edges.reserve(n);
3.  u = v = n + 2;
4.  intEdge e2 = 0;
5.  for (auto &edge : edges) {
6.      if(edge.second == v and edge.first == u)
7.          continue;
8.      e2++;
9.      u = edge.first;

```

```

10.     v = edge.second;
11.     unique_edges.push_back(make_pair(u, v));
12. }

```

④将上一步得到的边列表转换为图的紧凑表示，初始化相关数据结构

a. Edgelist 类用于存储和管理图的边列表。其数据结构定义如下。

```

1.     struct Edgelist {
2.     public:
3.         ul n;
4.         intEdge e;
5.         std::vector<edge> edges;
6.
7.         Edgelist(ul n, intEdge e);
8.         Edgelist(std::ifstream &file);
9.         Edgelist(const std::vector<std::pair<ul, ul>>& unique_edges, ul
            node_count, intEdge edge_count);
10.
11.         void sort_edges();
12.     };

```

其中， n 代表图中顶点总数， e 代表图中边的总数， $edges$ 是存储边的向量。

还可以对 Edgelist 类进行多种函数操作，包括：

Edgelist(ul n, intEdge e)

用于创建一个空的边列表，可以指定顶点数和边数以便后续添加边。

Edgelist(std::ifstream &file)

从输入文件中读取边并构建 Edgelist 对象，初始化 $edges$ 向量。

Edgelist(const std::vector<std::pair<ul, ul>>& unique_edges, ul node_count, intEdge edge_count)

从给定的边列表初始化一个新的 Edgelist 对象，初始化 $edges$ 向量，同时更新 n 和 e 的值。

sort_edges()

对 $edges$ 向量中的边进行排序，可以优化后续图算法的执行效率。

b. Adjlist 类用于存储和管理图的邻接表。其数据结构定义如下。

```

1.     struct Adjlist {
2.     public:
3.         bool directed;

```

```

4.     int edge_factor;
5.     int node_factor;
6.     ul n;
7.     intEdge e;
8.     std::vector<intEdge> cd;
9.     std::vector<ul> adj;
10.    std::function<ul(ul)> ranker;
11.    void ranker_reset();
12. };

```

其中， n 代表图中顶点总数， e 代表图中边的总数。`std::vector<intEdge> cd` 是累积度数数组，长度为 $n+1$ ，存储每个顶点的邻居在 `adj` 中的起始位置；`std::vector adj` 是所有顶点邻居的邻接列表，长度为 e ，存储所有顶点的邻居顶点。还可以对 `Adjlist` 类进行多种函数操作，包括：

`std::function<ul(ul)> ranker`

对顶点按照度数的高低进行排序。

`void ranker_reset()`

重置顶点排序函数。

基于上述两种数据结构，实现算法第④步的代码如下。

```

1.  Edgelist* h = NULL;
2.  Adjlist* g = NULL;
3.  srand ( unsigned ( time(0) ) );
4.  omp_set_num_threads(parallel_threads);
5.  h = new Edgelist(unique_edges, n, e2);

```

⑤对图的顶点排序进行初始化

- a. `order_identity` 函数返回一个一一对应、升序排列的顶点向量，实际上就是生成一个顺序排列的从 0 到 $n-1$ 的整数向量 `rank`（对 `rank` 进行初始化）。
- b. `rank` 存储了图中顶点的排序信息，用于后续算法优化和三角形计数。

```

1.  vector<ul> rank = order_identity(n);

```

⑥对图的顶点进行排序

循环遍历图中每个顶点 u 并计算其 dp 值。首先初始化 $dp[u] = 0$ ，然后对于 u 的每个邻居 v ，如果 v 的排序 `rank[v]` 大于 u 的排序 `rank[u]`，则增加 $dp[u]$ 的值。

这里实际上是根据 u 的排序信息来计算它的出度 dp （只针对比它排名更高的节点），排名越高的顶点越有可能参与构成一个三角形，因此在后续计算三角形个数时会优先考虑排名较高的顶点。

```
1. g = new Uadjlist(*h);
2.
3. vector<ul> dp(g->n, 0);
4. for(ul u=0; u<g->n; ++u) {
5.     dp[u] = 0;
6.     for(auto &v : g->neigh_iter(u))
7.         if(rank[v] > rank[u]) dp[u]++;
8. }
9.
10. g->soft_rank(rank, dp);
```

⑦三角形计数

在三角形计数阶段，我使用 OpenMP 并行化计算图中的三角形总数，充分利用多线程处理节点加速计算过程（对应 1.2.6 节优化点 4）。每个线程分别处理不同的节点，在各自的 `omp parallel` 区域内，对每个顶点进行以下操作：

- 根据顶点 u 的出边邻接列表，设置一个数组 `is_neighOut` 标记顶点 u 的所有邻居顶点。
- 对每个邻居 v ，将 `is_neighOut[v]` 设置为 `true`，表示 v 是 u 的邻居。然后进一步遍历 v 的邻居 w ，如果 w 也是 u 的邻居，则增加三角形计数 t （这是很显然的，因为三角形仅当邻居关系闭环时成立）。
- 处理完 u 的邻居后，将 `is_neighOut` 数组全部重置为 `false`，从而在下一次处理新顶点 u' 时，`is_neighOut` 数组不会保留过去顶点 u 的邻居信息，只会记录新顶点 u' 的邻居信息。这样避免了重复计入同一个三角形的问题。

```
1. ull t = 0, c = 0;
2. #pragma omp parallel reduction(+ : t, c)
3. {
4.     vector<bool> is_neighOut(g->n, false);
5.     #pragma omp for schedule(dynamic, 1)
6.     for(ul u=0; u<g->n; ++u) {
7.         for(ul i=g->cd[u]; i < g->cd[u] + dp[u]; ++i)
8.             is_neighOut[ g->adj[i] ] = true;
9.         for(ul i=g->cd[u]; i < g->cd[u] + dp[u]; ++i) {
10.             ul v = g->adj[i];
```



```

11.         for(ul j=g->cd[v]; j < g->cd[v] + dp[v]; ++j) {
12.             ul w = g->adj[j];
13.             if(is_neighOut[w])
14.                 t++;
15.                 c++;
16.         }
17.     }
18.
19.     for(ul i=g->cd[u]; i < g->cd[u] + dp[u]; ++i)
20.         is_neighOut[ g->adj[i] ] = false;
21.     }
22. }

```

1.2.3 优化点 1: 使用基于互斥锁的 OpenMP 多线程, 并行读取并处理输入数据

1.优化思路: 由于给定的数据集规模非常大, 如果采用单线程串行读入的方式, 则消耗时间过长。因此, 可以考虑引入多个线程, 它们各自负责读取和处理一部分数据, 并统一合并到最终的结果集合中。

具体流程包括以下三步:

- a. 首先声明每个线程的私有变量和共享变量。每个线程拥有两个私有变量用于暂存读取到的某条边的顶点 id; 所有线程共享三个变量, 包括边的集合 `edges`、全局顶点数量 `n` 和全局边的数量 `e`。
- b. 在并行区域内部, 每个线程从文件中读取一行数据, 然后解析出边的两个顶点 `u` 和 `v`。如果边是自环则跳过该边, 否则将其添加到 `edges` 中并更新节点数 `n` 和边数 `e`。这里使用了 `set` 作为边的存储结构, 相当于在存储边的时候已经完成了去重。
- c. 使用 `pragma omp critical` 创建一个临界区, 确保在任意时刻只有一个线程可以进入临界区访问共享变量, 防止多个线程同时修改它们导致的并发访问问题。

在实际测试中, 我设定的线程数为 16。

2.关键代码:

```

1.     #pragma omp parallel num_threads(parallel_threads) private(u, v)
        shared(edges, n, e)
2.     {
3.         string line;
4.         while (getline(file, line)) {
5.             stringstream ss(line);
6.             ss >> u >> v;

```

```

7.         if (u == v) continue; // self-loop
8.         if (u > v) swap(u, v);
9.
10.        // 使用互斥锁确保对共享数据的安全访问
11.        #pragma omp critical
12.        {
13.            edges.push_back(make_pair(u, v));
14.            n = max3(n, u, v);
15.            e++;
16.        }
17.    }
18. }

```

1.2.4 优化点 2：使用管道（Pipeline）进行通信优化，降低读写开销

1.优化思路：在对边信息完成去重后，一种方式是先将去重后的边输出到本地文件中，然后在后续计算阶段再次读入该文件。需要注意的是，在内存中进行 I/O 操作的时间消耗是巨大的，这样会带来大量的冗余存储开销。

如果直接将处理完的边保存在当前程序中，并直接以管道的形式传输给后续计算的函数，就可以省去一次写文件和一次读文件的时间开销，从而达到用空间换时间的效果。

2.关键代码：见 1.1.2 节步骤③。

1.2.5 优化点 3：使用 `__gnu_parallel::sort` 优化算法进行并行排序

1.优化思路：相比于经典的 `std::sort` 标准排序函数，`__gnu_parallel::sort` 是 GNU C++ 库的一个并行排序算法，它能利用多核处理器的并行能力，在排序时将数据分成多个部分并分配给多个线程同时处理。

该算法具有 Cache 友好的特点，因为它可以减少原始数据移动和访问的次数，降低了缓存竞争和内存访问延迟的影响。针对本实验中大规模边集合的排序，采用这一算法可以改善排序的效率。

2.关键代码：见 1.2.2 节步骤②。

1.2.6 优化点 4：基于 OpenMP 的多线程并行三角形计数

1.优化思路：与 1.2.3 节优化点 1 的想法一致，只是这里每个线程都各自负责统计一部分顶点和边所在的三角形个数，可以加速三角形计数的过程。

为了降低负载不均衡的影响，我使用 `schedule(dynamic, 1)` 进行动态调度，在节点度数差异较大的情况下能维持不同线程间的负载相对均衡。

2. 关键代码：

```
1.  #pragma omp parallel reduction(+ : t, c)
2.  {
3.      ...
4.      #pragma omp for schedule(dynamic, 1)
5.      ...
6.  }
```

1.2.7 算法流程图

综合上述算法思路和优化点，三角形计数算法的整体流程图如图 2 所示。

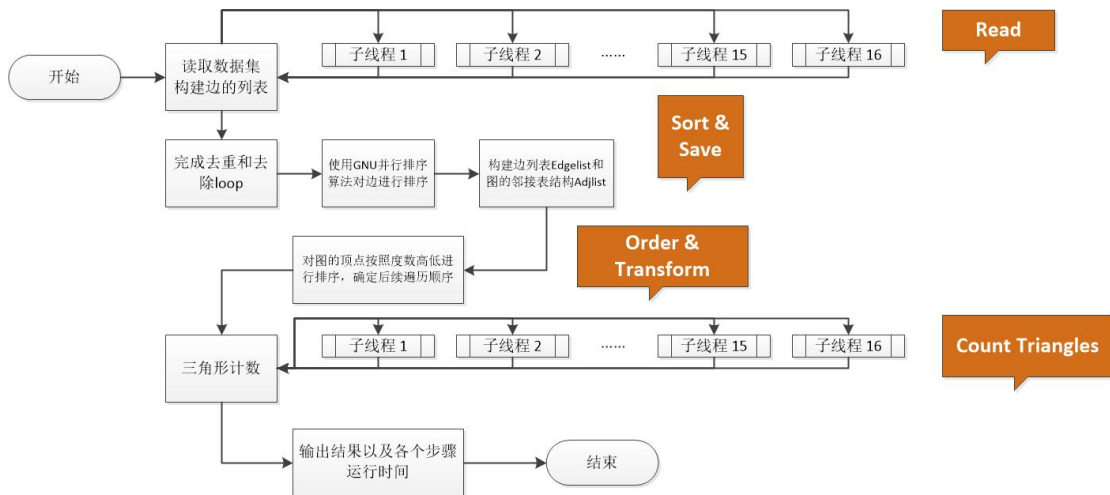


图 2：三角形计数算法流程图

1.2.8 阶段 3 实验结果与分析

1. 任务描述：编译程序并在给定的大数据集 soc-LiveJournal1 上运行，观察结果。

2. 实验过程：①首先切换到代码目录下，通过 `make` 命令编译，如图 3 所示。

```
make
```

```
U202115652@test-KVM:~/triangle_count$ make
g++ -std=c++14 triangle_count.o utils/edgelist.o utils/adjlist.o -Ofast -g -fopenmp -o triangle_count
```

图 3：程序编译通过

②输入下面的命令运行程序。

```
./triangle_count
```

3.实验结果：如图 4 所示。

①结果正确性


程序在大数据集 soc-LiveJournal1 上运行，输出三角形的总数为 **285730264**，与官方答案一致（图 5）。


②运行时间

程序总运行时间为 **7369ms (7.369s)**，相比优化前有大幅提高。相关数据如表 2 所示，据此可以计算出各阶段的加速比 $\frac{T_{\text{优化前}}}{T_{\text{优化后}}}$ 。

```
● U202115652@test-KVM:~/triangle_count$ ./triangle_count
(1) Read: 3608 ms
Nodes: 4847571
Edges: 68993773
Loops: 518382
(2) Sort: 626 ms
** Delete multi-edge & loop **
Saved 42851237 unique edges
** Running 16 parallel threads **
New Nodes: 4847571
New Edges: 42851237
(3) Save & Read: 811 ms
(4) Order: 12 ms
(5) Transform: 1675 ms
Triangle Count: 285730264
(6) Count Triangles: 637 ms
Total Time: 7369 ms
```

图 4：大数据集 soc-LiveJournal1 运行结果

**LiveJournal social network**

 **Dataset information**

LiveJournal is a free on-line community with almost 10 million members; a significant fraction of these members are highly active. (For example, roughly 300,000 update their content in any given 24-hour period.) LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong.

Dataset statistics	
Nodes	4847571
Edges	68993773
Nodes in largest WCC	4843953 (0.999)
Edges in largest WCC	68983820 (1.000)
Nodes in largest SCC	3828682 (0.790)
Edges in largest SCC	65825429 (0.954)
Average clustering coefficient	0.2742
Number of triangles	285730264

图 5：大数据集 soc-LiveJournal1 官方结果

表 2：优化前后各部分时间变化

阶段	优化前（ms）	优化后（ms）	加速比
Read	11775	3608	3.26
Sort	947	626	1.51
Save & Read	18972	811	23.39
Count Triangles	1056	637	1.66
总时间	32762	7369	4.45

可以看出加速比从 1.5 倍到 23.3 倍不等，证明上述优化效果明显。

1.3 实验小结

本次实验是针对大规模图数据进行三角形计数算法设计与性能优化。在当前的大数据时代下，图数据逐渐成为主流的数据组织形式，而图数据的三角形计数被广泛应用于图数据的特征描绘、社区结构检索、子图匹配等应用，具有极强的社会价值与现实意义。

在实验中，我首先设计了简单的三角形计数算法，包括：读入图数据、去除重边和 loop、采用暴力方法遍历全部顶点并计算三角形个数。不过很显然，暴力计算并不是最优解，在该算法之上还存在着大量潜在的优化点。我先后尝试了多种优化策略，其中效果显著的主要包括以下几项：（1）利用基于互斥锁的 OpenMP 多线程实现并行数据读取与处理，以及并行三角形计数的动态调度，有效减少了处理时间，避免了各线程之间负载不均衡的问题；（2）采用了管道（Pipeline）的通信方式，减少了中间文件的读写开销，直接传输处理后的数据，进一步节省了时间资源；（3）应用了 GNU 的并行排序算法优化了边的排序过程，有效降低了排序时间。上述优化策略为大规模图数据三角形高效计数提供了一种可行的解决方案，在大规模测试集 soc-LiveJournal1 上仅需要 7369ms 即可完成计算并输出正确结果。

通过本次实验，我深入理解了图计算系统的工作原理和性能优化机制，认识到在处理大规模数据时需要根据任务的特点来针对性地优化。通过结合实际情况，我引入了多线程并行、管道通信优化和高效排序算法等策略，在确保答案正确的前提下显著提升了程序的效率。作为一名大数据专业的学生，我的编程能力和调试技能得到了显著提高，在面对实际算法问题、不断迭代优化算法的过程中，我的分析问题、思考解决方案和持续改进的能力得到了有效的锻炼，为我未来的学习工作打下了坚实的基础。